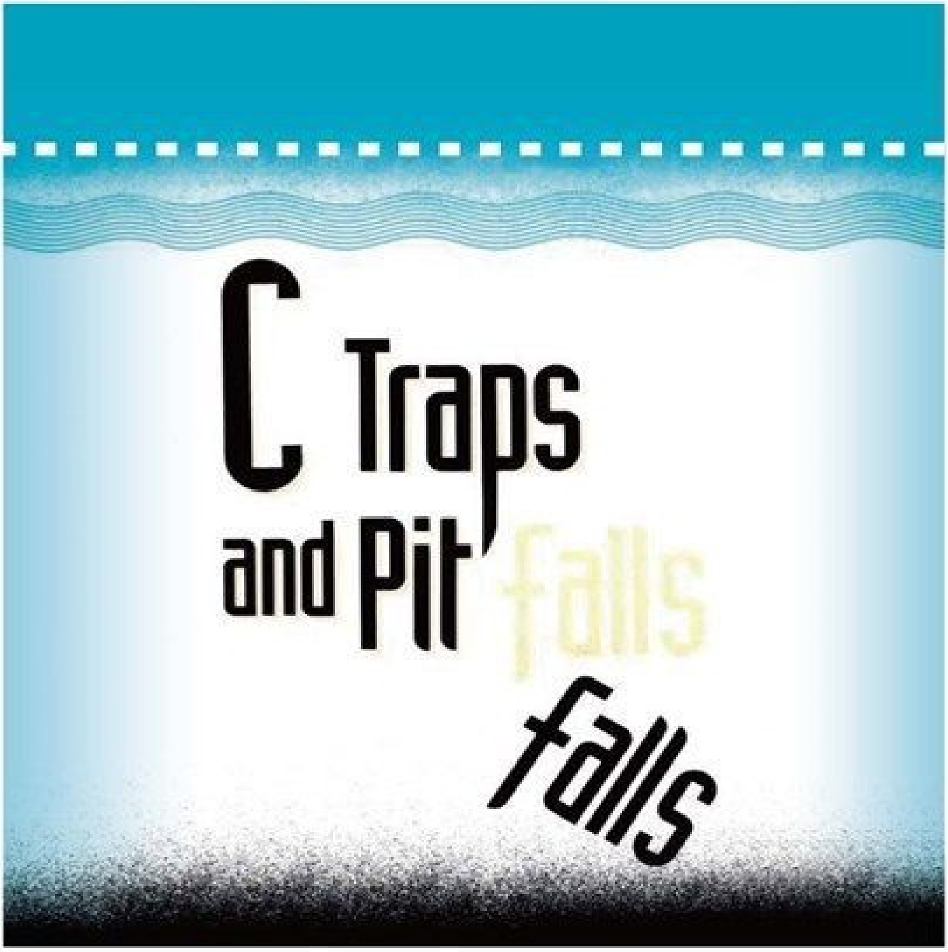
Addison Wesley

C和C++经典著作

C&C++ Classic



C陷阱与缺陷

C Traps and Pitfalls

〔美〕Andrew Koenig 著
高 巍 译
王 昕 审校

人民邮电出版社
POSTS & TELECOM PRESS

目　录

版权信息

书名：C陷阱与缺陷

ISBN：978-7-115-17179-5

- 著　　　Andrew Koenig

  译　　　高 巍

  审　　校　王 昕

  责任编辑　付　飞

- 人民邮电出版社出版发行　　北京市丰台区成寿寺路11号

  邮编　100164　电子邮件　315@ptpress.com.cn

  网址　http://www.ptpress.com.cn

- 读者服务热线：(010)81055410

  反盗版热线：(010)81055315

版权声明

Andrew Koenig：C Traps and Pitfalls

ISBN：0201179288

内容提要

作者以自己1985年在Bell实验室时发表的一篇论文为基础，结合自己的工作经验扩展成为这本对C程序员具有珍贵价值的经典著作。写作本书的出发点不是要批判C语言，而是要帮助C程序员绕过编程过程中的陷阱和障碍。

全书分为 8章，分别从词法分析、语法语义、连接、库函数、预处理器、可移植性缺陷等几个方面分析了C编程中可能遇到的问题。最后，作者用一章的篇幅给出了若干具有实用价值的建议。

本书适合有一定经验的C程序员阅读学习，即便你是C编程高手，本书也应该成为你的案头必备书籍。

Preface to the Chinese Edition

When I first wrote C Traps and Pitfalls, I never dreamed that it would still be in print 14 years later! I believe that the reason for this book's longevity is that it teaches some important lessons about C programming that are still not widely understood.

The aspects of C that invite mistakes are the same aspects that make it attractive for expert programmers. Accordingly, most people who set out to become C experts will make the same mistakes along the way——mistakes that will be there to be made as long as C continues to attract new programmers.

If you read a typical programming book, you will probably find that the author thinks that the most important part of becoming a good programmer is to learn as many details as possible about a particular language, library, or system. There is some truth in this notion of course, but it tells only part of the story. Details are easy to learn: All one needs is a reference book with a good index, and erhaps a more experienced colleague to point one in the right direction once in a while. It is much harder to understand the best ways of using what one already knows.

One way to gain such understanding is to learn what not to do.Programming languages, such as C, that are intended to be convenient for experts to use often invite misuse in ways that someone with enough experience can predict. By studying the mistakes that programmers make most often in such a language, one can not only avoid those mistakes, but one can also understand more deeply how the language works.

I am particularly pleased to learn about the Chinese translation of this book because the translation makes it available for the first time to such a large audience. If you read this book, I hope that it will help turn your frustration into happiness.

Andrew Koenig

Gillette, New Jersey, USA

October, 2002

18次印刷的奇迹

——经典C语言书籍*C Traps and Pitfalls*简介

如果有人问我，要想学好一门编程语言，应该阅读什么样的书籍呢？毫无疑问，在大多数场合下我都会向他推荐市面上最新出版的书籍。原因就是：以现在计算机领域内技术的发展速度，几乎是每隔一段时间，我们就需要对自己现有的知识进行更新，这样看来，使用一本比较新的书籍，里面的内容将会比较贴近当前技术的发展，因而也就能够让你更容易掌握你所要学的东西。

但有一本讲述C语言的书，自出版以来，历经14载，它一直都被各个书评站点（或书评人）列入“重点推荐”的清单中。尤为夸张的是，14年来，在它的18次印刷版本中，除去第二次印刷稍微修改过一些问题，以后的16次印刷，我们居然发现它的内容没有丝毫的变更！！！对于技术书籍，我想其精确性与权威性也算是奇迹了吧。

这就是Andrew Koenig给我们带来的*C Traps and Pitfalls*（中文书名：《C陷阱与缺陷》）。在C/C++领域中，Andy（Andrew的昵称）的名字对于每个人来说绝对是如雷贯耳。作为一个知名的专栏作者，Andy（和他那位同样大名鼎鼎的妻子——Barbara Moo）已经在各类杂志上面发表了上百篇的杂志文章，给很多人在技术进步的道路上带来了极大的帮助。ACCU的Francis Glassborow对他的评价是：Andy是世界上最出色的几位C++专家之一。

本书是Andy的第一本技术书籍，其原始素材来自于Andy在1986所提交的同名的技术报告。在书中，作者针对C程序在编译、链接的过程中可能碰到的种种问题以及编译、运行环境对程序可能带来的影响等，列出了许多值得我们注意的地方。按照作者本人的观点，前人碰到过的问题来现身说法，可以帮助你避免那些一而再，再而三出现在你的程序中的问题。由于是以实例来描述作者（以及他人）所碰到过的具体问题，因此，此书少去了许多空洞无味的说教，虽然本书篇幅不大（原书正文只有区区147页），但实际上，它的每个小节，每一段都蕴含着作者（以及他人）大量的经验教训，都值得我们去仔细琢磨，经常温习。为此，Francis Glassborow说到：从我了解C语言开始，我就将它时时放

在手边，经常翻阅。作者自己也在书中毫不谦虚地说："如果你是一个程序员在开发中经常用到C语言，这本书应该成为你案头必备书籍。即使你已经是一个C语言的专家级程序员，仍然有必要拥有一本"。事实上，Andy并没有吹嘘，就书中所列出的种种问题，我本人也不止一次在自己的程序（也包括别人的程序）中发现它们的踪迹，而且有些问题出现得还极为频繁。这使我不禁就想到，要是我们能够早一些看到这本书上提及的问题，那岂不是可以省去很多开发时的除错时间……

可能有人会有疑问：从书名来看，它是一本讲述C语言的书籍，那么对于C++的学习者来说，它是不是也同样有价值呢？另外，现在C语言的最新ISO/ANSI标准文档C99都已经制订出来了，而作为一本在C89之前出版的C语言书籍，它的作用是否还和以前一样大呢？答案是肯定的。本书英文版连续18次印刷的事实就是有力的证明。实际上，C++在那个层面上和C的区别并不大，在C程序中常犯的错误通常在C++程序中也经常得以重现，因此，从这个角度来说，C语言中的陷阱也常常就是C++语言中的陷阱。此外，虽然最新的C99对于以前的K & R C有了一些变化，但在较低层次（如词法、语法）上，它们几乎是没有差别的。因此，对于本书中所有问题的讨论，几乎都可以适用于最新的ISO/ANSI C。

现在，人民邮电出版社翻译出版*C Traps and Pitfalls*一书，无疑是献给C和C++程序员的一份厚礼。我本人很荣幸能够担任此书的技术审校，为这本书中文版的出版尽一点绵薄的心力。感谢译者的辛勤劳动，也感谢出版社能够给我这样的机会！希望本书能够为你的学习带来一些帮助。

王昕

2002年8月

中文版序

我动笔写作《C缺陷与陷阱》时，可没想到14年后这本书仍然在印行！它之所以历久不衰，我想，可能是书中道出了C语言编程中一些重要的经验教训。就是到今天，这些教训也还没有广为人知。

C语言中那些容易导致人犯错误的特性，往往也正是编程老手们为之吸引的特性。因此，大多数程序员在成长为C编程高手的道路上，犯过的错误真是惊人地相似！只要C语言还能继续感召新的程序员投身其中，这些错误就还会一犯再犯。

大家通常读到的程序设计书籍中，那些作者总是认为，要成为一个优秀的程序员，最重要的无非是学习一种特定程序语言、函数库或者操作系统的细节，而且多多益善。当然，这种观念不无道理，但也有偏颇之处。其实，掌握细节并不难，一本索引丰富完备的参考书就已经足矣；最多，可能还需要一位稍有经验的同事不时从旁点拨，指明方向。难的是那些我们已经了解的东西，如何“运用之妙，存乎一心”。

学习哪些是不应该做的，倒不失为一条领悟运用之道的路子。程序设计语言，就比如说C吧，其中那些让精于编程者觉得称心应手之处，也格外容易误用；而经验丰富的老手，甚至可以如有“先见之明”般，指出他们误用的方式。研究一种语言中程序员容易犯错之处，不但可以“前车之覆，后车之鉴”，还能使我们更谙熟这种语言的深层运作机制。

知悉本书中文版即出，将面对更为广大的中国读者，我尤为欣喜。如果您正在读这本书，我真挚地希望，它能对您有所裨益，为您释疑解惑，让您体会编程之乐。

Andrew Koenig

美国新泽西州吉列

2002年10月

作者简介



Andrew Koenig

AT&T大规模程序研发部（前贝尔实验室）成员。他从1986年开始从事C语言的研究，1977年加入贝尔实验室。他编写了一些早期的类库，并在1988年组织召开了第一个相当规模的C++会议。在ISO/ANSI C++委员会成立的1989年，他就加入了该委员会，并一直担任项目编辑。他已经发表了C++方面的100多篇论文，在Addsion-Wesley出版了*C Trap and Pitfalls,*（《C陷阱与缺陷》）和*Ruminations on C++*（《C++沉思录》，人民邮电出版社）还应邀到世界各地演讲。

Andrew Koenig不仅有着多年的C++开发、研究和教学经验，而且还亲身参与了C++的演化和变革，对C++的变化和发展起到重要的影响。

前言

对于经验丰富的行家而言，得心应手的工具在初学时的困难程度往往要超过那些容易上手的工具。刚刚接触飞机驾驶的学员，初航时总是谨小慎微，只敢沿着海岸线来回飞行，等他们稍有经验就会明白这样的飞行其实是一件多么轻松的事。初学骑自行车的新手，可能觉得后轮两侧的辅助轮很有帮助，但一旦熟练过后，就会发现它们很是碍手碍脚。

这种情况对程序设计语言也是一样。任何一种程序设计语言，总存在一些语言特性，很可能会给还没有完全熟悉它们的人带来麻烦。令人吃惊的是，这些特性虽然因程序设计语言的不同而异，但对于特定的一种语言，几乎每个程序员都是在同样的一些特性上犯过错误、吃过苦头！因此，作者也就萌生了将这些程序员易犯错误的特性加以收集、整理的最初念头。

我第一次尝试收集这类问题是在1977年。当时，在华盛顿特区举行的一次SHARE（IBM 大型机用户组）会议上，我作了一次题为“PL/I中的问题与‘陷阱’”的发言。作此发言时，我刚从哥伦比亚大学调至AT&T的贝尔实验室，在哥伦比亚大学我们主要的开发语言是PL/I，而贝尔实验室中主要的开发语言却是C。在贝尔实验室工作的10年间，我积累了丰富的经验，深谙C程序员（也包括我本人）在开发时如果一知半解将会遇到多少麻烦。

1985年，我开始收集有关C语言的此类问题，并在年底将结果整理后作为一篇内部论文发表。这篇论文所引发的回应却大大出乎我的意料，共有2 000多人向贝尔实验室的图书馆索取该论文的副本！我由此确信有必要将该论文的内容进一步扩充，于是就写成了现在读者所看到的这本书。

本书是什么

本书力图通过揭示一般程序员，甚至是经验老道的职业程序员，如何在编程中犯错误、摔跟头，以提倡和鼓励预防性的程序设计。这些错误实际上一旦被程序员真正认识和理解，并不难避免。因此，本书阐述的重点不是一般原则，而是一个个具体的例子。

如果你是一个程序员并且开发中真正用到C语言来解决复杂问题，这本书应该成为你的案头必备书籍。即使你已经是一个C语言的专家级程序员，仍然有必要拥有这本书，很多读过本书早期手稿的专业C程序员常常感叹："就在上星期我还遇到这样一个Bug！"如果你正在教授C语言课程，本书毫无疑问应该成为你向学生推荐的首选补充阅读材料。

本书不是什么

本书不是对C语言的批评。程序员无论使用何种程序设计语言，都有可能遇到麻烦。本书浓缩了作者长达10年的C语言开发经验，集中阐述了C语言中各种问题和"陷阱"，目的是希望程序员读者能够从中吸取我本人以及我所见过的其他人所犯错误的经验教训。

本书不是一本"烹饪菜谱"。我们不能希望可以通过详尽的指导说明来完全避免错误。如果可行的话，那么所有的交通事故都可以通过在路旁刷上"小心驾驶"的标语来杜绝。对一般人而言最有效的学习方式是从感性的、活生生的事例中学习，比如自己的亲身经历或者他人的经验教训。而且，哪怕只是明白了一种特定的错误是如何可能发生的，就已经在将来避免该错误的路上迈了一大步。

本书并不打算教你如何用C语言编程（见Kernighan 和 Ritchie： *The C Programming Language*， 第2版，Prentice-Hall，1988），也不是一本C语言参考手册（见Harbison 和 Steele：*C*：*A Reference Manual*，第2版，Prentice-Hall，1987）。本书未提及数据结构与算法（见Van Wyk：*Data Structures And C Programs*，Addison-Wesley，1988），仅仅简略介绍了可移植性（见Horton： *How To Write Portable Programs In C*，Prentice-Hall，1989）和操作系统接口（见Kernighan 和 Pike： *The Unix Programming Environment*，Prentice-Hall，1984）。本书中所涉及的问题均来自编程实践，适当作了简化（如果希望读到一些"挖空心思"设计出来，专门让你绞尽脑汁的C语言难题，见Feuer： *The C Puzzle Book*， Prentice-Hall，1982）。本书既不是一本字典也不是一本百科全书，我力图使其精简短小，以鼓励读者能够阅读全书。

读者的参与和贡献

可以肯定，我遗漏了某些值得注意的问题。如果你发现了一个C语言问题而本书又未提及，请通过Addison-Wesley出版社与我联系。在本书的下一版中，我很有可能引用你的发现，并且向你致谢。

关于**ANSI C**

在我写作本书时，ANSI C标准尚未最后定案。严格地说，在ANSI委员会完成其工作之前，“ANSI C”的提法从技术上而言是不正确的。而实际上，ANSI标准化工作大体已经尘埃落定，本书中提及的有关ANSI C标准内容基本上不可能有所变动。很多C编译器甚至已经实现了大部分ANSI委员会所考虑的对C语言的许多重大改进。

毋需担心你使用的C编译器并不支持书中出现的ANSI标准函数语法，它并不会妨碍你理解例子中真正重要的内容，而且书中提及的程序员易犯错误其实与何种版本的C编译器并无太大关系。

致谢

本书中问题的收集整理工作绝非一人之力可以完成。以下诸位都向我指出过C语言中的特定问题，他们是Steve Bellovin（6.3节），Mark Brader（1.1节），Luca Cardelli（4.4节），Larry Cipriani（2.3节），Guy Harris and Steve Johnson（2.2节），Phil Karn（2.2节），Dave Kristol（7.5节），George W. Leach（1.1节），Doug McIlroy（2.3节），Barbara Moo（7.2节），Rob Pike（1.1节），Jim Reeds（3.6节），Dennis Ritchie（2.2节），Janet Sirkis（5.2节），Richard Stevens（2.5节），Bjarne Stroustrup（2.3节），Ephraim Vishnaic（1.4节），以及一位自愿要求隐去姓名者（2.3节）。为简短起见，对于同一个问题此处仅仅列出了第一位向我指出该问题的人。我认为这些错误绝不是凭空臆造出来的，而且即使是，我想也没有人愿意承认。至少这些错误我本人几乎都犯过，而且有的还不止犯一次。

在书稿编辑方面许多有用的建议来自Steve Bellovin， Jim Coplien，Marc Donner， Jon Forrest，Brian Kernighan，Doug McIlroy，Barbara Moo，Rob Murray，Bob Richton，Dennis Ritchie，Jonathan Shapiro，以及一些未透露姓名的审阅者。Lee McMahon与Ed Sitar为我指出了早期手稿中的许多录入错误，使我避免了一旦成书后将要遇到的很多尴尬。Dave Prosser为我指明了许多ANSI C中的细微之处。Brian Kernighan提供了极有价值的排版工具和帮助。

与Addison-Wesley出版社合作是一件愉快的事情，感谢Jim DeWolf，Mary Dyer，Lorraine Ferrier，Katherine Harutunian，Marshall Henrichs，Debbie Lafferty，Keith Wollman，和Helen Wythe。当然，他

们也从一些并不为我所知的人们那里得到了帮助，使本书最终得以出版，我在此也一并致谢。

我需要特别感谢AT&T贝尔实验室的管理层，他们开明的态度和支持使我得以写作本书，包括Steve Chappell，Bob Factor，Wayne Hunt，Rob Murray，Will Smith，Dan Stanzione和Eric Sumner。

本书书名受到Robert Sheckley的科幻小说选集的启发，其书名是*The People Trap and Other Pitfalls*， *Snares*， *Devices and Delusions* （*as well as Two Sniggles and a Contrivance*）（1968年由Dell Books出版）。

第0章　导读

我的第一个计算机程序写于1966年，是用Fortran语言开发的。该程序需要完成的任务是计算并打印输出10 000以内的所有Fibonacci数，也就是一个包括1，1，2，3，5，8，13，21，……等元素的数列，其中第2个数字之后的每个数字都是前两个数字之和。当然，写程序代码很难第一次就顺利通过编译：

```
  I = 0
  J = 0
  K = 1
1 PRINT 10,K
  I = J
  J = K
  K = I + J
  IF (K - 10000) 1, 1, 2
2 CALL EXIT
10 FORMAT(I10)
```

Fortran程序员很容易发现上面这段代码遗漏了一个END语句。当我添上END语句之后，程序还是不能通过编译，编译器的错误消息也让人迷惑不解：ERROR 6。

通过仔细查阅编译器参考手册中对错误消息的说明，我最后终于明白了问题所在：我使用的Fortran编译器不能处理4位数以上的整型常量。将上面这段代码中的10000改为9999，程序就顺利通过了编译。

我的第一个C程序写于1977年。当然，第一次还是没有得到正确结果：

```
#include <stdio.h>

main()
{
          printf("Hello world");
}
```

这段代码虽然在编译时一次通过，但是，程序执行的结果看上去却有点奇怪。终端输出差不多就是下面这样：

```
% cc prog.c
% a.out
```

```
Hello world%
```

这里的%字符是系统提示符，操作系统用它来提示用户输入。因为在程序中没有写明“Hello world”消息之后应该换行，所以系统提示符%直接出现在输出的“Hello world”消息之后。这个程序中还有一个更加难以察觉的错误，将在本书的3.10节加以讨论。

上面提到的两个程序中所出现的错误，是有着实质区别的两种不同类型的错误。在Fortran程序的例子中出现了两个错误，但是这两个错误都能够被编译器检测出来。而C程序的例子从技术上说是正确的，至少从计算机的角度来看它没有错误。因此，C程序顺利通过了编译，没有报告任何警告或错误消息。计算机严格地按照我写明的程序代码来执行，但结果却并不是我真正希望得到的。

本书所要集中讨论的是第二类问题，也就是程序并没有按照程序员所期待的方式执行。更进一步，本书的讨论限定在C语言程序中可能产生这类错误的方式。例如，考虑下面这段代码：

```
int i;
int a[N];
for (i = 0; i <= N; i++)
        a[i] = 0;
```

这段代码的作用是初始化一个N元数组，但是在很多C编译器中，它将会陷入一个死循环！本书的3.6节讨论了为什么会这样的原因。

程序设计错误实际上反映的是程序与程序员对该程序的“心智模式”[①]两者的相异之处。从程序错误的本性而言，我们很难给它们进行恰当的分类。对一个程序错误可以从不同层面采用不同方式进行考察，根据程序错误与考察程序的方式之间的相关性，我尝试着对程序错误进行了划分。

> 译注[①]：心智模式（mental model）在彼得·圣吉的《第五项修炼——学习型组织的艺术与实务》（上海三联书店，1998年第2版）中也有提到，被解释为“人们深植心中，对于周遭世界如何运作的看法和行为”。Howard Gardner在研究认知科学的一本著作《心灵的新科学》（_The Mind's New Science_）中认为，人们的心智模式决定了人们如何认识周遭世界。《列子》一书中有个典型的故事，说有个人遗失了一把斧头，他怀疑是邻居孩子偷的，暗中观察他的行为，怎么看怎么像偷斧头的人；后来他在自己家中找到了遗失的斧头，再碰到邻居的孩子时，怎么看也不像会是偷他斧头的人了。

从较低的层面考察，程序是由符号（token）序列所组成的，正如

一本书是由一个一个单词所组成的一样。将程序分解成符号的过程，称为“词法分析”。第1章考察在程序被词法分析器分解成各个符号的过程中可能出现的问题。

组成程序的这些符号，又可以看成是语句和声明的序列，就好像一本书可以看成是由单词进一步结合而成的句子所组成的集合。无论是对于书而言，还是对于程序而言，符号或者单词如何组成更大的单元（对于前者是语句和声明，对于后者是句子）的语法细节最终决定了语义。如果没有正确理解这些语法细节，将会出现怎样的错误呢？第2章就此进行了讨论。

第3章处理有关语义误解的问题：即程序员的本意是希望表示某种事物，而实际表示的却是另外一种事物。在这一章中我们假定程序员对词法细节和语法细节的理解没有问题，因此着重讨论语义细节。

第4章注意到这样一个事实：C程序经常是由若干个部分组成，它们分别进行编译，最后再整合起来。这个过程称为“连接”，是程序和其支持环境之间关系的一部分。

程序的支持环境包括某组库函数（library routine）。虽然严格说来库函数并不是语言的一部分，但是库函数对任何一个有用的程序都非常重要。特别地，有些库函数几乎每个C程序都要用到。对这些库函数的误用可以说是五花八门，因此值得在第5章中专门讨论。

在第6章，我们还注意到，由于C预处理器的介入，实际运行的程序并不是最初编写的程序。虽然不同预处理器的实现存在或多或少的差异，但是大部分特性是各种预处理器都支持的。第6章讨论了与这些特性有关的有用内容。

第7章讨论了可移植性问题，也就是为什么在一个实现平台上能够运行的程序却无法在另一个平台上运行。当牵涉到可移植性时，哪怕是非常简单的类似整数的算术运算这样的事情，其困难程度也常常会出人意料。

第8章提供了有关预防性程序设计的一些建议，还给出了其他章节的练习解答。

最后，附录中讨论了3个常用的却普遍地被误解的库函数。

练习**0-1.**　你是否愿意购买一个返修率很高的厂家所生产的汽车？如果厂家声明它已经做出了改进，你的态度是否会改变？用户为你找出程序中的Bug，你真正损失的是什么？

练习**0-2.**　修建一个100英尺长的护栏，护栏的栏杆之间相距10英尺，你需要多少根栏杆？

练习**0-3.**　在烹饪时你是否失手用菜刀切伤过自己的手？怎样改进菜刀使得使用更安全？你是否愿意使用这样一把经过改良的菜刀？

第1章　词法“陷阱”

当我们阅读一个句子时，我们并不去考虑组成这个句子的单词中单个字母的含义，而是把单词作为一个整体来理解。确实，字母本身并没有什么意义，我们总是将字母组成单词，然后给单词赋予一定的意义。

对于用C语言或其他语言编写的程序，道理也是一样的。程序中的单个字符孤立来看并没有什么意义，只有结合上下文才有意义。因此，在p->s = "->";这个语句中，两处出现的'-'字符的意义大相径庭。更精确地说，上式中出现的两个'-'字符分别是不同符号的组成部分：第一个'-'字符是符号->的组成部分，而第二个'-'字符是一个字符串的组成部分。此外，符号->的含义与组成该符号的字符'-'或字符'>'的含义也完全不同。

术语“符号”（token）指的是程序的一个基本组成单元，其作用相当于一个句子中的单词。从某种意义上说，一个单词无论出现在哪个句子，它代表的意思都是一样的，是一个表义的基本单元。与此类似，符号就是程序中的一个基本信息单元。而组成符号的字符序列就不同，同一组字符序列在某个上下文环境中属于一个符号，而在另一个上下文环境中可能属于完全不同的另一个符号。

> 译注：
>
> 如上面的字符'-'和字符'>'组成的字符序列->，在不同的上下文环境中，一个代表->运算符，一个代表字符串"->"。

编译器中负责将程序分解为一个一个符号的部分，一般称为“词法分析器”。

再看下面一个例子，语句：

```
if (x > big) big = x;
```

这个语句的第一个符号是C语言的关键字if，紧接着下一个符号是左括号，再下一个符号是标识符x，再下一个是大于号，再下一个是标识符big，依次类推。在C语言中，符号之间的空白（包括空格符、制表符或换行符）将被忽略，因此上面的语句还可以写成：

```
if
(
x
>
big
)
big
=
x
;
```

本章将探讨符号和组成符号的字符间的关系，以及有关符号含义的一些常见误解。

1.1 =不同于==

由Algol派生而来的大多数程序设计语言，例如Pascal和Ada，使用符号:=作为赋值运算符，符号=作为比较运算符。而C语言使用的是另一种表示法，符号=作为赋值运算，符号= =作为比较。一般而言，赋值运算相对于比较运算出现得更频繁，因此字符数较少的符号=就被赋予了更常用的含义——赋值操作。此外，在C语言中赋值符号被作为一种操作符对待，因而重复进行赋值操作（如a=b=c）可以很容易地书写，并且赋值操作还可以被嵌入到更大的表达式中。

这种使用上的便利性可能导致一个潜在的问题：当程序员本意是作比较运算时，却可能无意中误写成了赋值运算。比如下例，该语句本意似乎是要检查x是否等于y：

```
if (x = y)
      break;
```

而实际上是将y的值赋给了x，然后检查该值是否为零。再看下面一个例子，本例中循环语句的本意是跳过文件中的空格符、制表符和换行号：

```
while (c = ' ' || c == '\t' || c == '\n')
      c = getc (f);
```

由于程序员在比较字符' '和变量c时，误将比较运算符= =写成了赋值运算符=。因为赋值运算符=的优先级要低于逻辑运算符 || ，因此实际上是将以下表达式的值赋给了c：

```
' ' || c == '\t' || c == '\n'
```

因为 ' ' 不等于零（' ' 的ASCII码值为32），那么无论变量c此前为何值，上述表达式求值的结果都是1，因此循环将一直进行下去直到整个文件结束。文件结束之后循环是否还会进行下去，这取决于getc库函数的具体实现，在文件指针到达文件结尾之后是否还允许继续读取字符。如果允许继续读取字符，那么循环将一直进行，从而成为一个死循环。

某些C编译器在发现形如e1 = e2的表达式出现在循环语句的条件判断部分时，会给出警告消息以提醒程序员。当确实需要对变量进行赋值

并检查该变量的新值是否为0时，为了避免来自该类编译器的警告，我们不应该简单关闭警告选项，而应该显式地进行比较。也就是说，下例

```
if (x = y)
        foo();
```

应该写作：

```
if ((x = y) != 0)
        foo();
```

这种写法也使得代码的意图一目了然。至于为什么要用括号把x = y括起来，本书的2.2节将讨论这个问题。

前面一直谈的是把比较运算误写成赋值运算的情形，另一方面，如果把赋值运算误写成比较运算，同样会造成混淆：

```
if ((filedesc == open(argv[i], 0)) < 0)
        error();
```

在本例中，如果函数open执行成功，将返回0或者正数；而如果函数open执行失败，将返回-1。上面这段代码的本意是将函数open的返回值存储在变量filedesc之中，然后通过比较变量filedesc是否小于0来检查函数open是否执行成功。但是，此处的= =本应是=。而按照上面代码中的写法，实际进行的操作是比较函数open的返回值与变量filedesc，然后检查比较的结果是否小于0。因为比较运算符= =的结果只可能是0或1，永远不可能小于0，所以函数error()将没有机会被调用。如果代码被执行，似乎一切正常，除了变量filedesc的值不再是函数open的返回值（事实上，甚至完全与函数open无关）。某些编译器在遇到这种情况时，会警告与0比较无效。但是，作为程序员不能指望靠编译器来提醒，毕竟警告消息可以被忽略，而且并不是所有编译器都具备这样的功能。

1.2 &和 | 不同于&&和 ||

很多其他语言都使用=作为比较运算符，因此很容易误将赋值运算符=写成比较运算符= =。同样地，将按位运算符&与逻辑运算符&&，或者将按位运算符 | 与逻辑运算符 || 调换，也是很容易犯的错误。特别是C语言中按位与运算符&和按位或运算符 | ，与某些其他语言中的按位与运算符和按位或运算符在表现形式上完全不同（如Pascal语言中分别是and和or），更容易让程序员因为受到其他语言的影响而犯错。关于这些运算符精确含义的讨论见本书的3.8节。

1.3 词法分析中的“贪心法”

C语言的某些符号，例如/ 、* 、和=，只有一个字符长，称为单字符符号。而C语言中的其他符号，例如/ *和 = = ，以及标识符，包括了多个字符，称为多字符符号。当C编译器读入一个字符'/'后又跟了一个字符'*'，那么编译器就必须做出判断：是将其作为两个分别的符号对待，还是合起来作为一个符号对待。C语言对这个问题的解决方案可以归纳为一个很简单的规则：每一个符号应该包含尽可能多的字符。也就是说，编译器将程序分解成符号的方法是，从左到右一个字符一个字符地读入，如果该字符可能组成一个符号，那么再读入下一个字符，判断已经读入的两个字符组成的字符串是否可能是一个符号的组成部分；如果可能，继续读入下一个字符，重复上述判断，直到读入的字符组成的字符串已不再可能组成一个有意义的符号。这个处理策略有时被称为“贪心法”，或者，更口语化一点，称为“大嘴法”。Kernighan与Ritchie对这个方法的表述如下，“如果（编译器的）输入流截止至某个字符之前都已经被分解为一个个符号，那么下一个符号将包括从该字符之后可能组成一个符号的最长字符串。”

需要注意的是，除了字符串与字符常量，符号的中间不能嵌有空白（空格符、制表符和换行符）。例如，= =是单个符号，而= = 则是两个符号，下面的表达式

```
a---b
```

与表达式

```
a -- - b
```

的含义相同，而与

```
a - -- b
```

的含义不同。同样地，如果/是为判断下一个符号而读入的第一个字符，而/之后紧接着*，那么无论上下文如何，这两个字符都将被当作一个符号/ *，表示一段注释的开始。

根据代码中注释的意思，下面的语句的本意似乎是用x除以p所指向

的值，把所得的商再赋给y：

```
y = x/*p      /* p指向除数*/;
```

而实际上，/*被编译器理解为一段注释的开始，编译器将不断地读入字符，直到*/出现为止。也就是说，该语句直接将x的值赋给y，根本不会顾及到后面出现的p。将上面的语句重写如下：

```
y = x / *p    /* p指向除数 */;
```

或者更加清楚一点，写作：

```
y = x/(*p)    /* p指向除数 */;
```

这样得到的实际效果才是语句注释所表示的原意。

诸如此类的准二义性（near-ambiguity）问题，在有的上下文环境中还有可能招致麻烦。例如，老版本的C语言中允许使用=+来代表现在+=的含义。这种老版本的C编译器会将

```
a=-1;
```

理解为下面的语句

```
a =- 1;
```

亦即

```
a = a - 1;
```

因此，如果程序员的原意是

```
a = -1;
```

那么所得结果将使其大吃一惊。

另一方面，尽管/*看上去像一段注释的开始，在下例中这种老版本的编译器会将

```
a=/*b;
```

当作

```
a =/ *b ;
```

这种老版本的编译器还会将复合赋值视为两个符号，因而可以毫无疑问地处理

```
a >> = 1;
```

而一个严格的ANSI C编译器则会报错。

1.4　整型常量

如果一个整型常量的第一个字符是数字0，那么该常量将被视作八进制数。因此，10与010的含义截然不同。此外，许多C编译器会把8和9也作为八进制数字处理。这种多少有点奇怪的处理方式来自八进制数的定义。例如，0195的含义是$1\times8^2+9\times8^1+5\times8^0$，也就是141（十进制）或者0215（八进制）。我们当然不建议这种用法，ANSI C标准也禁止这种用法。

需要注意这种情况，有时候在上下文中为了格式对齐的需要，可能无意中将十进制数写成了八进制数，例如：

```
struct {
        int part_number;
        char *description;
}parttab[] = {
        046,   "left-handed widget"     ,
        047,   "right-handed widget"    ,
        125,    "frammis"
};
```

1.5 字符与字符串

C语言中的单引号和双引号含义迥异，在某些情况下如果把两者弄混，编译器并不会检测报错，从而在运行时产生难以预料的结果。

用单引号引起的一个字符实际上代表一个整数，整数值对应于该字符在编译器采用的字符集中的序列值。因此，对于采用ASCII字符集的编译器而言，'a'的含义与0141（八进制）或者97（十进制）严格一致。

用双引号引起的字符串，代表的却是一个指向无名数组起始字符的指针，该数组被双引号之间的字符以及一个额外的二进制值为零的字符'\0'初始化。

下面的这个语句：

```
printf ("Hello world\n");
```

与

```
char hello[] = {'H', 'e', 'l', 'l', 'o', ' ',
          'w', 'o', 'r', 'l', 'd', '\n', 0};
printf (hello);
```

是等效的。

因为用单引号括起的一个字符代表一个整数，而用双引号括起的一个字符代表一个指针，如果两者混用，那么编译器的类型检查功能将会检测到这样的错误。例如：

```
char *slash = '/';
```

在编译时将会生成一条错误消息，因为'/'并不是一个字符指针。然而，某些C编译器对函数参数并不进行类型检查，特别是对printf函数的参数。因此，如果用

```
printf('\n');
```

来代替正确的

```
printf("\n");
```

则会在程序运行的时候产生难以预料的错误，而不会给出编译器诊断信息。本书的4.4节还详细讨论了其他情形。

> 译注：
>
> 现在的编译器一般能够检测到在函数调用时混用单引号和双引号的情形。

整型数（一般为16位或32位）的存储空间可以容纳多个字符（一般为8位），因此有的C编译器允许在一个字符常量（以及字符串常量）中包括多个字符。也就是说，用'yes'代替"yes"不会被该编译器检测到。后者（即"yes"）的含义是“依次包含'y'、'e'、's'以及空字符'\0'的4个连续内存单元的首地址”。前者（即'yes'）的含义并没有准确地进行定义，但大多数C编译器理解为，“一个整数值，由'y'、'e'、's'所代表的整数值按照特定编译器实现中定义的方式组合得到”。因此，这两者如果在数值上有什么相似之处，也完全是一种巧合而已。

> 译注：
>
> 在Borland C++ v5.5和LCC v3.6中采取的做法是，忽略多余的字符，最后的整数值即第一个字符的整数值；而在Visual C++ 6.0和GCC v2.95中采取的做法是，依次用后一个字符覆盖前一个字符，最后得到的整数值即最后一个字符的整数值。

练习**1-1.**　某些C编译器允许嵌套注释。请写一个测试程序，要求：无论是对允许嵌套注释的编译器，还是对不允许嵌套注释的编译器，该程序都能正常通过编译（无错误消息出现），但是这两种情况下程序执行的结果却不相同。

> 提示：
>
> 在用双引号括起的字符串中，注释符 / *属于字符串的一部分，而在注释中出现的双引号" "又属于注释的一部分。

练习**1-2.**　如果由你来实现一个C编译器，你是否会允许嵌套注释？如果你使用的C编译器允许嵌套注释，你会用到编译器的这一特性吗？你对第二个问题的回答是否会影响到你对第一个问题的回答？

练习**1-3.**　为什么n-->0的含义是n-- > 0，而不是n- -> 0？

练习**1-4.**　a+++++b的含义是什么？

第2章　语法“陷阱”

要理解一个C程序，仅仅理解组成该程序的符号是不够的。程序员还必须理解这些符号是如何组合成声明、表达式、语句和程序的。虽然这些组合方式的定义都很完备，几乎无懈可击，但有时这些定义与人们的直觉相悖，或者容易引起混淆。本章将讨论一些用法和意义与我们想当然的认识不一致的语法结构。

2.1 理解函数声明

有一次，一个程序员与我交谈一个问题。他当时正在编写一个独立运行于某种微处理器上的C程序。当计算机启动时，硬件将调用首地址为0位置的子例程。

为了模拟开机启动时的情形，我们必须设计出一个C语句，以显式调用该子例程。经过一段时间的思考，我们最后得到的语句如下：

```
(*(void(*)())0)();
```

像这样的表达式恐怕会令每个C程序员的内心都“不寒而栗”。然而，他们大可不必对此望而生畏，因为构造这类表达式其实只有一条简单的规则：按照使用的方式来声明。

任何C变量的声明都由两部分组成：类型以及一组类似表达式的声明符（declarator）。声明符从表面上看与表达式有些类似，对它求值应该返回一个声明中给定类型的结果。最简单的声明符就是单个变量，如：

```
float f, g;
```

这个声明的含义是：当对其求值时，表达式f和g的类型为浮点数类型（float）。因为声明符与表达式的相似，所以我们也可以在声明符中任意使用括号：

```
float ((f));
```

这个声明的含义是：当对其求值时，((f))的类型为浮点类型，由此可以推知，f也是浮点类型。

同样的逻辑也适用于函数和指针类型的声明，例如：

```
float ff();
```

这个声明的含义是：表达式ff()求值结果是一个浮点数，也就是说，ff是一个返回值为浮点类型的函数。类似地，

```
float *pf;
```

这个声明的含义是*pf是一个浮点数，也就是说，pf是一个指向浮点数的指针。

以上这些形式在声明中还可以组合起来，就像在表达式中进行组合一样。因此，

```
float *g(), (*h)();
```

表示*g()与(*h)()是浮点表达式。因为()结合优先级高于*，*g()也就是*(g())：g是一个函数，该函数的返回值类型为指向浮点数的指针。同理，可以得出h是一个函数指针，h所指向函数的返回值为浮点类型。

一旦我们知道了如何声明一个给定类型的变量，那么该类型的类型转换符就很容易得到了：只需要把声明中的变量名和声明末尾的分号去掉，再将剩余的部分用一个括号整个“封装”起来即可。例如，因为下面的声明：

```
float (*h)();
```

表示h是一个指向返回值为浮点类型的函数的指针，因此，

```
(float (*)())
```

表示一个“指向返回值为浮点类型的函数的指针”的类型转换符。

拥有了这些预备知识，我们现在可以分两步来分析表达式 (*(void(*)())0)() 。

第一步，假定变量fp是一个函数指针，那么如何调用fp所指向的函数呢？调用方法如下：

```
(*fp)();
```

因为fp是一个函数指针，那么*fp就是该指针所指向的函数，所以(*fp)()就是调用该函数的方式。ANSI C标准允许程序员将上式简写为fp()，但是一定要记住这种写法只是一种简写形式。

在表达式(*fp)()中，*fp两侧的括号非常重要，因为函数运算符()的优先级高于单目运算符*。如果*fp两侧没有括号，那么*fp()实际上与*(fp())的含义完全一致，ANSI C把它作为*((*fp)())的简写形式。

现在，剩下的问题就只是找到一个恰当的表达式来替换fp。我们将在分析的第二步来解决这个问题。如果C编译器能够理解我们大脑中对于类型的认识，那么我们可以这样写：

```
(*0)();
```

上式并不能生效，因为运算符*必须要一个指针来做操作数。而且，这个指针还应该是一个函数指针，这样经运算符*作用后的结果才能作为函数被调用。因此，在上式中必须对0作类型转换，转换后的类型可以大致描述为：“指向返回值为void类型的函数的指针”。

如果fp是一个指向返回值为void类型的函数的指针，那么(*fp)()的值为void，fp的声明如下：

```
void (*fp)();
```

因此，我们可以用下式来完成调用存储位置为0的子例程：

```
void (*fp)();
(*fp)();
```

> 译注：
>
> 此处作者假设fp默认初始化为0，这种写法不宜提倡。

这种写法的代价是多声明了一个“哑”变量。

但是，我们一旦知道如何声明一个变量，也就自然知道如何对一个常数进行类型转换，将其转型为该变量的类型：只需要在变量声明中将变量名去掉即可。因此，将常数0转型为“指向返回值为void的函数的指针”类型，可以这样写：

```
(void (*)())0
```

因此，我们可以用(void (*)())0来替换fp，从而得到：

```
(*(void (*)())0)();
```

末尾的分号使得表达式成为一个语句。

在我当初解决这个问题的时候，C语言中还没有typedef声明。尽管不用typedef来解决这个问题对剖析本例的细节而言是一个很好的方式，但无疑使用typedef能够使表述更加清晰：

```
typedef void (*funcptr)();
(*(funcptr)0)();
```

这个棘手的例子并不是孤立的，还有一些C程序员经常遇到的问题，实际上和这个例子是同一个类型的。例如，考虑signal库函数，在包括该函数的C编译器实现中，signal函数接受两个参数：一个是代表需要“被捕获”的特定signal的整数值；另一个是指向用户提供的函数的指针，该函数用于处理“捕获到”的特定signal，返回值类型为void。我们将会在本书5.5节详细讨论该函数。

一般情况下，程序员并不主动声明signal函数，而是直接使用系统头文件signal.h中的声明。那么，在头文件signal.h中，signal函数是如何声明的呢？

首先，让我们从用户定义的信号处理函数开始考虑，这无疑是最容易解决的。该函数可以定义如下：

```
void sigfunc(int n){
            /* 特定信号处理部分*/
}
```

函数sigfunc的参数是一个代表特定信号的整数值，此处我们暂时忽略它。

上面假设的函数体定义了sigfunc函数，因而sigfunc函数的声明可以如下：

```
void sigfunc(int )；
```

现在假定我们希望声明一个指向sigfunc函数的指针变量，不妨命名为sfp。因为sfp指向sigfunc函数，则*sfp就代表了sigfunc函数，因此*sfp可以被调用。又假定sig是一个整数，则(*sfp)(sig)的值为void类型，因此

我们可以如下声明sfp：

```
void (*sfp)(int);
```

因为signal函数的返回值类型与sfp的返回类型一样，上式也就声明了signal函数，我们可以如下声明signal函数：

```
void (*signal(something))(int);
```

此处的something代表了signal函数的参数类型，我们还需要进一步了解如何声明它们。上面声明可以这样理解：传递适当的参数以调用signal函数，对signal函数返回值（为函数指针类型）解除引用（dereference），然后传递一个整型参数调用解除引用后所得函数，最后返回值为void类型。因此，signal函数的返回值是一个指向返回值为void类型的函数的指针。

那么，signal函数的参数又是如何呢？signal函数接受两个参数：一个整型的信号编号，以及一个指向用户定义的信号处理函数的指针。我们此前已经定义了指向用户定义的信号处理函数的指针sfp：

```
void (*sfp)(int);
```

sfp 的类型可以通过将上面的声明中的sfp去掉而得到，即void (*)(int)。此外，signal函数的返回值是一个指向调用前的用户定义信号处理函数的指针，这个指针的类型与sfp指针类型一致。因此，我们可以如下声明signal函数：

```
void (*signal(int, void(*)(int)))(int);
```

同样地，使用typedef可以简化上面的函数声明：

```
typedef void (*HANDLER)(int);
HANDLER signal(int, HANDLER);
```

2.2 运算符的优先级问题

假设存在一个已定义的常量FLAG，FLAG是一个整数，且该整数值的二进制表示中只有某一位是1，其余各位均为0，亦即该整数是2的某次幂。如果对于整型变量flags，我们需要判断它在常量FLAG为1的那一位上是否同样也为1，通常可以这样写：

```
if (flags & FLAG) …
```

上式的含义对大多数C程序员来说是显而易见的：if语句判断括号内表达式的值是否为0。考虑到可读性，如果对表达式的值是否为0的判断能够显式地加以说明，无疑使得代码自身就起到了注释该段代码意图的作用。其写法如下，

```
if (flags & FLAG != 0) …
```

这个语句现在虽然更好懂了，但却是一个错误的语句。因为!=运算符的优先级要高于 & 运算符，所以上式实际上被解释为：

```
if (flags & (FLAG != 0) ) …
```

因此，除了FLAG恰好为1的情形，FLAG为其他数时这个式子都是错误的。

又假设hi和low是两个整数，它们的值介于0到15之间，如果r是一个8位整数，且r的低4位与low各位上的数一致，而r的高4位与hi各位上的数一致。很自然会想到要这样写：

```
r = hi<<4 + low;
```

但是很不幸，这样写是错误的。加法运算的优先级要比移位运算的优先级高，因此本例实际上相当于：

```
r = hi<< (4 + low);
```

对于这种情况，有两种更正方法：第一种方法是加括号；第二种方法意识到问题出在程序员混淆了算术运算与逻辑运算，但这种方法牵涉

到的移位运算与逻辑运算的相对优先级就更加不是那么明显。两种方法如下：

```
r = （hi<<4） + low;      //法1：加括号
r = hi<<4 | low;         //法2：将原来的加号改为按位逻辑或
```

用添加括号的方法虽然可以完全避免这类问题，但是表达式中有了太多的括号反而不容易理解。因此，记住C语言中运算符的优先级是有益的。

遗憾的是，运算符优先级有15个之多，因此记住它们并不是一件容易的事。完整的C语言运算符优先级表如表2-1所示。

表**2-1**　　**C**语言运算符优先级表（由上至下，优先级依次递减）

运 算 符	结 合 性
() [] -> .	自左向右
! ~ ++ -- - (type) * & sizeof	自右至左
* / %	自左向右
+ -	自左向右
<< >>	自左向右
< <= > >=	自左向右
== !=	自左向右
&	自左向右
^	自左向右

		自左向右
&&	自左向右
||	自左向右
?:	自右至左
assignments	自右至左
,	自左向右

如果把这些运算符恰当分组，并且理解了各组运算符之间的相对优先级，那么这张表其实不难记住。

优先级最高者其实并不是真正意义上的运算符，包括：数组下标、函数调用操作符各结构成员选择操作符。它们都是自左于右结合，因此a.b.c的含义是(a.b).c，而不是a.(b.c)。

单目运算符的优先级仅次于前述运算符。在所有的真正意义上的运算符中，它们的优先级最高。因为函数调用的优先级要高于单目运算符的优先级，所以如果p是一个函数指针，要调用p所指向的函数，必须这样写：(*p)()。如果写成*p()，编译器会解释成*(p())。类型转换也是单目运算符，它的优先级和其他单目运算符的优先级一样。单目运算符是自右至左结合，因此*p++会被编译器解释成*(p++)，即取指针p所指向的对象，然后将p递增1；而不是(*p)++，即取指针p所指向的对象，然后将该对象递增1。本书3.7节还进一步指出了p++的含义有时会出人意料。

优先级比单目运算符要低的，接下来就是双目运算符。在双目运算符中，算术运算符的优先级最高，移位运算符次之，关系运算符再次之，接着是逻辑运算符，赋值运算符，最后是条件运算符。

译注：

原书如此，条件运算符实际应为三目运算符。

我们需要记住的最重要的两点是：

1．任何一个逻辑运算符的优先级低于任何一个关系运算符。

2．移位运算符的优先级比算术运算符要低，但是比关系运算符要高。

属于同一类型的各个运算符之间的相对优先级，理解起来一般没有什么困难。乘法、除法和求余优先级相同，加法、减法的优先级相同，两个移位运算符的优先级也相同。1/2*a的含义是(1/2)*a，而不是1/(2*a)，这一点也许会让某些人吃惊，其实在这方面C语言与Fortran语言、Pascal语言以及其他程序设计语言之间的行为表现并无差别。

但是，6个关系运算符的优先级并不相同，这一点或许让人感到有些吃惊。运算符==和!=的优先级要低于其他关系运算符的优先级。因此，如果我们要比较a与b的相对大小顺序是否和c与d的相对大小顺序一样，就可以这样写：

```
a < b == c < d
```

任何两个逻辑运算符都具有不同的优先级。所有的按位运算符优先级要比顺序运算符的优先级高，每个“与”运算符要比相应的“或”运算符优先级高，而按位异或运算符（^运算符）的优先级介于按位与运算符和按位或运算符之间。

这些运算符的优先顺序是由于历史原因形成的。B语言是C语言的“祖先”，B语言中的逻辑运算符大致相当于C语言中的&和 | 运算符。虽然这些运算符从定义上而言是按位操作的，但是当它们出现在条件语句的上下文中时，B语言的编译器会将它们作为相当于现在C语言中的&&和 || 运算符处理。而到了C语言中，这两种不同的用法被区分开来，从兼容性的角度来考虑，如果对它们优先顺序的改变过大将是一件危险的事。

在本节到现在为止提及的所有运算符中，三目条件运算符优先级最低。这就允许我们在三目条件运算符的条件表达式中包括关系运算符的逻辑组合，例如：

```
tax_rate = income>40000 && residency<5 ? 3.5: 2.0;
```

本例其实还揭示了：赋值运算符的优先级低于条件运算符的优先级是有意义的。此外，所有的赋值运算符的优先级是一样的，而且它们的结合方式是从右到左，因此，

```
home_score = visitor_score = 0;
```

与下面两条语句所表达的意思是相同的：

```
visitor_score = 0;
home_score = visitor_score;
```

在所有的运算符中，逗号运算符的优先级最低。这一点很容易记住，因为逗号运算符常用于在需要一个表达式而不是一条语句的情形下替换作为语句结束标志的分号。逗号运算符在宏定义中特别有用，这一点在本书的6.3节还会进一步讨论。

在涉及到赋值运算符时，经常会引起优先级的混淆。考虑下面的这个例子，例子中循环语句的本意是复制一个文件到另一个文件：

```
while (c=getc(in) != EOF)
        putc(c,out);
```

在while语句的表达式中，c似乎是首先被赋予函数getc(in)的返回值，然后与EOF比较是否到达文件结尾以便决定是否终止循环。然而，由于赋值运算符的优先级要低于任何一个比较运算符，因此c的值实际上是函数getc(in)的返回值与EOF比较的结果。此处函数getc(in)的返回值只是一个临时变量，在与EOF比较后就被“丢弃”了。因此，最后得到的文件“副本”中只包括了一组二进制值为1的字节流。

上例实际应该写成：

```
while ((c=getc(in)) != EOF)
         putc(c,out);
```

如果表达式再复杂一点，这类错误就很难被察觉。例如，本书第4章章首提及的lint程序的一个版本，在发布时包括了下面一行错误代码：

```
if( (t=BTYPE(pt1->aty)==STRTY) || t==UNIONTY){
```

这行代码本意是首先赋值给t，然后判断t是否等于STRTY或者UNIONTY。实际的结果却大相径庭：根据BTYPE(pt1->aty)的值是否等于STRTY，t的取值或者为1或者为0；如果t取值为0，还将进一步与UNIONTY比较。

2.3 注意作为语句结束标志的分号